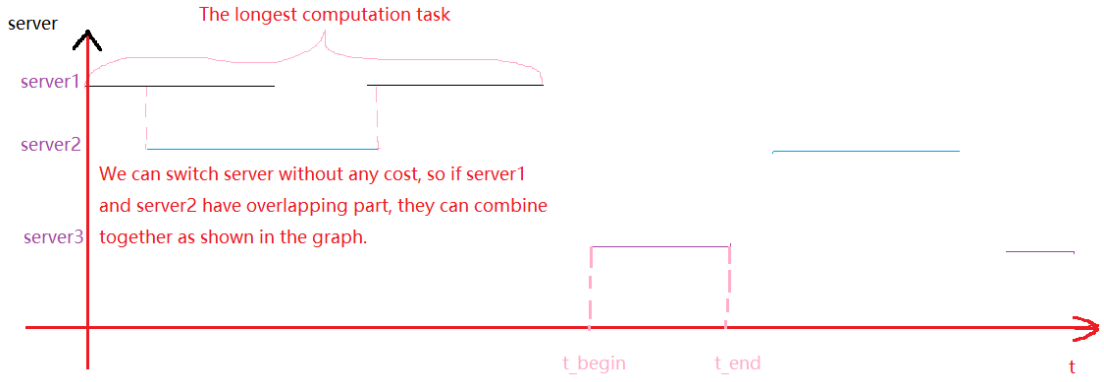
Project 3. Arrangement of Computation Tasks

December 25, 2020

Chapter 1: Introduction

In the computing center, there are a large number of servers running at different time in a day. Now that if we have the information of the running time of each of the server in hand, we can find a way to tell the longest computation task the computing center could run, and find the total amount of valid starting time for a given computation task, if it is runnable. What deserves your attention is that the task can only be run continuously to finish all its computation, which means we cannot stop it and restart it again to continue. Meanwhile, a computation task is allowed to switch from one server to another at any particular time with no cost, if both servers are active at that time.

The graph below may give you a rough sketch.



This project introduce an algorithm which is commonly used in the arrangement of the large quantity of servers in the computing center. When the computing task is going to be run by the center, we can judge whether it can run and if it can run, when it can run? How many choices can we have? In fact, this project shows the very first step in the arrangement of

the computing center. Only when we figure out the longest computing task as well as the running time choices of each task can we do a better arrangement and make as many programs as possible to work. In conclusion, from my personal point of view, this project is very meaningful extremely stretchable.

Chapter 2: Algorithm Specification

2.1 Structure

2.1.1 struct TimeNode.

As each input file contains one test case. Each case starts with two positive integers N and K. Then N lines follow, each gives a record in the format:

```
Server_number hh:mm:ss
```

The input example in PTA is as follows.

```
12 7
jh007bd 18:00:01
zd00001 11:30:08
db8888a 13:00:00
za3q625 23:59:50
za133ch 13:00:00
zd00001 04:09:59
za3q625 11:42:01
za3q625 06:30:50
za3q625 23:55:00
za133ch 17:11:22
jh007bd 23:07:01
db8888a 11:35:50
```

Where hh, mm, ss represent the hour, minute, and second of the record, with the earliest time being 00:00:00 and latest time being 23:59:59. In a

bid to store the hour, minute, and second of each record, structure TimeNode is designed:

```
typedef struct TimeNode time;
struct TimeNode
{
    int hour, minute, second;
};
```

2.1.2 struct TimeLine.

This structure is used to store the start and end time of a server. Define the point TimeLine* as timeline to point to the next data of the start time and end time of a server. In this part, the name of the server is not important, because what we only care about is the length of each computing task and whether overlapping part exist between two different record. So, the name of the server is unnecessary to record.

The detail of the structure is as follows.

```
typedef struct TimeLine *timeline;
struct TimeLine
{
    int hour1, minute1, second1, hour2, minute2, second2;
    timeline next;
};
```

2.1.3 struct ServerData.

This structure is designed to store the information of each server, containing the name of the server, server_number[7], the time_point,

which has been discussed earlier, and the integer count_time, which is to count the number of the time_point. For example, if we input “za3q625 11:42:01; za3q625 06:30:50; za3q625 23:55:00”, the value of the count_time of the point “za3q625” is 3.

Details are shown below.

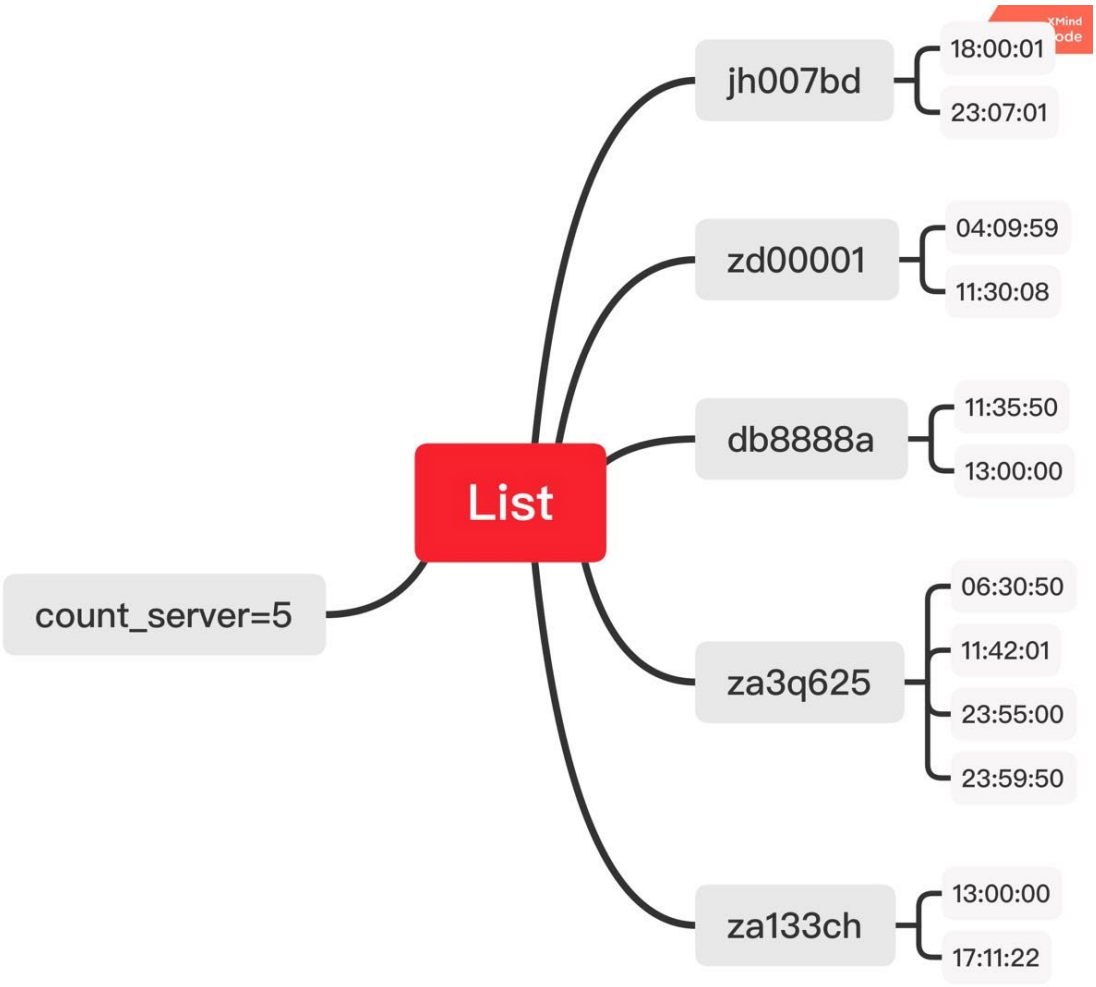
```
typedef struct ServerData Server;
struct ServerData
{
    /*
    The number of the server;
    */
    char server_number[7];

    time time_point[20000];
    int count_time;
};
```

2.1.4 struct top.

The structure list is the pointer of the structure top. And the structure top is used to store all the information, that’s why it has the name “top”.

The server_info, with type Server, stored the record of the time of different servers, while count_server is to count the total number of the different servers.

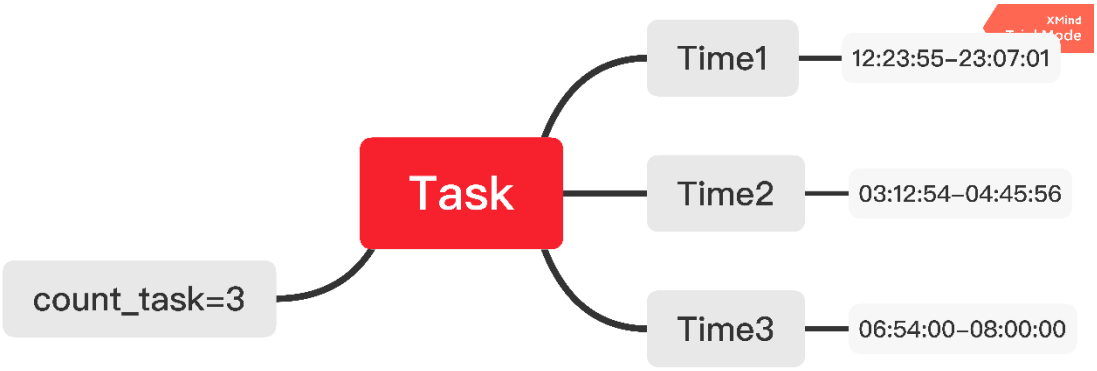
The following mind map may make clear how the structure list store the data. The data is based on the input example shown in PTA.




```
typedef struct top *list;
struct top
{
    Server server_info[10000];
    int count_server;
};
```

2.1.5 struct compute_list.

The structure compute_list is used to store the task after all the information is stored in the structure list. The following mind map shows a possible example, which may give you a clear sketch.



The source code of the structure is here:

```
typedef struct compute_list task;
struct compute_list
{
    timeline compute_task;
    int count_task;
};
```

2.2 Algorithm

2.2.1 list init_list(void).

Init the list, the number of the server is initialized as 0.

```
list init_list(void)
{
    list a = (list)malloc(sizeof(struct top));
    a->count_server = 0;
    return a;
}
```

2.2.2 int add_time(timeline a, time t1, time t2, int count).

This function is to add the time point to the list properly. Here is the algorithm. First, if timeline a is NULL, that is to say, count is 0. We just

simply add the time point to the head of the timeline a without any judgement.

```
If count == 0:
tmp->time = time_at_present;
a = tmp;
a->next = NULL;
```

If else, then further discussion is needed. We need a while loop to traverse the list, when we find a time point exist in the list that is the mother's gathering of the new time point, which means the time point doesn't need to be added, all we need to do is break the while loop.

```
If (new time point ⊂ old time point):
Free(new time point);
Break;
```

Else, if the new time point doesn't have an overlapping part with the old time point, we just need to continue the while loop without any action.

```
If(new time point ∉ old time point):
Tmp = tmp->next;
Continue;
```

Else, if the new time point has an overlapping part with the old time point, we need to change the value of time to combine the two time points together.

Last, we have to delete the extra time point if necessary.

```
Timeline obj = New(TimeLine);
Obj = obj->next; … … until obj->next == the new time point;
Obj->next = obj->next;
```

```
Free(the new time point);
```

2.2.3 list Read(int n).

Read the information of each server and store in the general list.

```
list Read(int n)
{
    general_list = New(general_list);
    for I in range(n)
    {
        Input(name[7], hour, minute, second);
        if (name[7] not in list)
            insert(new time point);
    }
    return general_list;
}
```

2.2.4 task AddTheData(list general_list).

The function AddTheData is designed to add the time point to the type task. The main algorithm is use a double-while-loop in order to read all the time point.

Here is pseudo-code of the function AddTheData():

```
For I in range(general_list->count_server):
    For j in range(general_list->server_info[i].count_time):
        Add_time();
```

2.2.5 int string_compare(list a, char b[], int hour, int minute, int second).

The function is able to decide whether the two input strings are the

same and return 1 if they are they are the same, return 0 if not. Meanwhile, the function can add the time point after the exist record if the strings are the same, because the same strings represent the same server, the data of the same server should be stored together.

```
int string_compare(list a, char b[], int hour, int minute, int second)
{
    if (!a->count_server) return 0;
    int i, flag = 0;
    for (i = 0; i < a->count_server; i++)
    {
        if (!strcmp(b, a->server_info[i].server_number))
        {
            flag = 1;
            add(a, hour, minute ,second, i);
            break;
        }
    }
    if (flag) return 1;
    return 0;
}
```

2.2.6 void add(list a, int hour, int minute, int second, int n).

This function has the capability to sort the time point in increasing order by insertion sort.

```
{
    if (a->server_info[n].count_time != 0)
```

```
    {
        for (i = a->server_info[n].count_time-1; i >= 0; i--)
        {
          if          (compare_time(a->server_info[n].time_point[i].hour,
          a->server_info[n].time_point[i].minute,
          a->server_info[n].time_point[i].second,       hour,       minute,
          second))
             {
                      a->server_info[n].time_point[i+1]=
                      a->server_info[n].time_point[i];
             }
             else break;
        }
    }
    a->server_info[n].time_point[i+1].hour = hour;
    a->server_info[n].time_point[i+1].minute = minute;
    a->server_info[n].time_point[i+1].second = second;
    a->server_info[n].count_time++;
}
```

2.2.7 int compute_second(int hour1, int minute1, int second1, int hour2, int minute2, int second2).

Compute the total seconds of a time line and return the total value in

integer type.

```
Total seconds =
(hour2*3600+minute2*60+second2)-
(hour1*3600+minute1*60+second1).
```

2.2.8 int compare_time(int hour1, int minute1, int second1, int hour2, int minute2, int second2).

The function compare the value of two time, h1:m1:s1 & h2:m2:s2, if h1:m1:s1 > h2:m2:s2, return 1; else, return 0.

```
{
    if (hour1 > hour2)
        return 1;
    else if (hour1 < hour2)
        return 0;
    else if (hour1 == hour2)
    {
        if (minute1 > minute2)
            return 1;
        else if (minute1 < minute2)
            return 0;
        else if (minute1 == minute2)
        {
            if (second1 > second2)
                return 1;
            else return 0;
        }
    }
}
```

2.2.9 void insert(list a, char name[], int hour, int minute, int second).

The function insert the time point to the list. If the list is empty, add the time point to the server_info[0]:

```
if (!a->count_server)
    {
        a->server_info[0].count_time = 0;
        a->count_server = 0;
        for (i = 0; i < 7; i++)
            a->server_info[0].server_number[i] = name[i];
        a->server_info[0].time_point[0].hour = hour;
        a->server_info[0].time_point[0].minute = minute;
        a->server_info[0].time_point[0].second = second;
        a->server_info[0].count_time++;
    }
```

Else, if the list is not empty, add the time list to the proper place.

```
a->server_info[a->count_server].count_time = 0;
for (i = 0; i < 7; i++)
a->server_info[a->count_server].server_number[i] = name[i];
a->server_info[a->count_server].time_point[0].hour = hour;
a->server_info[a->count_server].time_point[0].minute = minute;
a->server_info[a->count_server].time_point[0].second = second;
a->server_info[a->count_server].count_time++;
```

At last, after the insert, the number of the server is plus one, so we need to do “a->count_server++” at the end of the function.

Chapter 3: Testing Results

3.1 Simple test result with computation time larger than 23:59:59.

The purpose of this test is to judge whether the output is 0 when the computation time is bigger than the biggest time length, namely 23:59:59.

input	Output
2 1	86399

abcdefg 00:00:00 abcdefg 23:59:59 25:00:00	0

3.2 When the computing time is the largest and legal, 23:59:59.

The expected output is 1, the purpose of this test is to find whether it is correct or not.

input	output
2 1 abcdefg 00:00:00 abcdefg 23:59:59 23:59:59	86399 1

It is functioning well.

3.3 When the computing time is the smallest.

As time <= 0 is illegal, so the smallest number is 1. The purpose of this test is to find whether the program is functioning well in this part.

input	output
2 1 abcdefg 00:00:00 abcdefg 23:59:59 00:00:01	86399 86399

3.4 Comprehensive test from PTA.

The input is from PTA, it is a comprehensive test, by use this test file, we can judge whether the insertion sort of the time line is functioning well, and we can also judge whether the task time is larger than the max time, and return “0”.

input	Output
12 7	31801

jh007bd 18:00:01 zd00001 11:30:08 db8888a 13:00:00 za3q625 23:59:50 za133ch 13:00:00 zd00001 04:09:59 za3q625 11:42:01 za3q625 06:30:50 za3q625 23:55:00 za133ch 17:11:22 jh007bd 23:07:01 db8888a 11:35:50 08:30:01 12:23:42 05:10:00 04:11:21 00:04:10 05:06:59 00:05:11	1201 0 13202 20063 64597 13385 64373

3.5 Smallest computation task with smallest time.

The purpose of this case is to test whether the smallest computation task with smallest time is functioning well. The output is “1 1”, as expected.

input	output
2 1 sn00001 00:00:01 sn00001 00:00:02 00:00:01	1 1

3.6 The smallest and 2nd smallest computation task with smallest time.

For the 2nd smallest computation task, the choice is “0”, as expected. The purpose is to examine the function of the algorithm further. Meanwhile, the different servers have the same open/close time, so it offers us an opportunity to find out whether it has a bug in this situation.

input	output
6 2 sn00001 00:00:01 sn00001 00:00:02 sn00002 00:00:01 sn00002 00:00:02 sn00003 00:00:01 sn00003 00:00:02 00:00:01 00:00:02	1 1 0

3.7 Same server has more than one time point with the same time.

This is supposed to be an illegal input as the same server can’t search the computing server at the same time more than once. So the expected output is “0 0 0”. The purpose of this case is to find whether the program is able to solve the wrong input properly.

input	output
4 2 sn00001 00:00:01 sn00001 00:00:02 sn00001 00:00:01 sn00001 00:00:02 00:00:01 00:30:30	0 0 0

input	output
6 2 sn00001 00:00:01 sn00001 00:00:02 sn00001 00:00:01 sn00001 00:00:02 sn00001 00:00:01 sn00001 00:00:02 00:00:01 00:00:02	1 1 0

3.8 Start from 0 with no overlapping part.

When the input starts from 0 with no overlapping part, the result is correct as expected.

input	output
4 3 aaaaaaa 00:00:00 aaaaaaa 00:00:05 bbbbbbb 00:00:10 bbbbbbb 00:00:15 00:00:03 00:00:06 00:00:15	5 6 0 0

3.9 Start from 0, with overlapping part.

The purpose is to check whether the program is functioning well when facing the problem that the time points are adjacent or have overlapping part. The result is correct as expected.

input	output

8 7 aaaaaaa 00:00:00 aaaaaaa 00:00:08 bbbbbbb 00:00:08 bbbbbbb 00:00:16 ccccccc 00:00:16 ccccccc 00:00:46 ddddddd 00:00:50 ddddddd 00:00:51 00:00:08 00:00:09 00:00:01 00:00:02 00:00:15 00:00:30 23:59:59	30 25 22 47 43 16 1 0

3.10 Start from 0, and 23:59:59 is included

Input	output
4 7 aaaaaa 00:00:00 aaaaaa 00:00:20 bbbbbbb 23:59:50 bbbbbbb 23:59:59 00:00:19 00:00:20 00:00:21 00:00:09 00:00:10	20 2 1 0 13 11 10 9

00:00:11 00:00:12	

3.11 Overlapping, and all the time is included

input	output
10 7 aaaaaa 00:00:00 aaaaaa 00:00:20 bbbbbbb 23:59:50 bbbbbbb 23:59:59 ccccccc 00:00:10 ccccccc 18:00:30 ccccccc 20:31:31 ccccccc 23:59:55 ddddddd 17:00:01 ddddddd 21:40:40 01:00:00 02:00:00 06:00:00 12:00:00 18:00:00 23:00:00 23:59:59	86399 82800 79200 64800 43200 21600 3600 1

3.12 Big data 1 (you can find it in the file "big data")

The purpose of this big data is to check whether the illegal big data input can output the correct answer. The output is correct as expected.

input	output

20000 1 sn00000 00:00:00 sn00000 00:00:01 sn00001 00:00:00 sn00001 00:00:01 sn00002 00:00:00 sn00002 00:00:01 sn00003 00:00:00 sn00003 00:00:01 … … 00:00:02	1 0

3.13 Big data 2 ***(you can find it in the file "big data")***

Input	output
20000 80000 sn00000 00:00:00 sn00000 00:00:01 sn00001 00:00:00 sn00001 00:00:01 sn00002 00:00:00 sn00002 00:00:01 … … 00:00:01 00:00:01 … …	1 6667 6667 6667 … …

3.14 Big data 3 ***(you can find it in the file "big data")***

Input	output
20000 80000	1

sn00000 00:00:00	0
sn00000 00:00:01	0
sn00001 00:00:00	0
sn00001 00:00:01	… …
sn00002 00:00:00	
sn00002 00:00:01	
… …	
00:00:02	
00:00:03	
00:00:04	
00:00:05	
… …	
22:13:17	
22:13:18	
22:13:19	
22:13:20	
22:13:21	

3.15 Big data 4 *(you can find it in the file "big data")*

Input	output
20000 80000	1
sn10000 00:00:00	10000
sn10000 00:00:01	10000
sn10000 00:00:02	10000
sn10000 00:00:03	… …
sn10000 00:00:04	
sn10000 00:00:05	
sn10000 00:00:06	
sn10000 00:00:07	

sn10000 00:00:08 sn10000 00:00:09 … … 00:00:01 00:00:01 00:00:01 … …	

Chapter 4: Analysis and Comments

4.1 Analysis

4.1.1 Algorithm 1

The algorithm of storing the N records is to use insertion sort, which means we need to do the double-while loop to do it, so the time complexity is $O(N^2)$. There are N/2 records to be stored, so the space complexity is O(N). The algorithm of calculating the choices of the starting time of a task uses the double while loop to traverse of the time line. The worst time complexity is $O(N^2)$ if there is no overlapping part , and the best time complexity is O(N) if every two elements have overlapping part. So the average time complexity for this part is O((N+2N+3N+…+(N-1)N+N^2)/N) = $O\left(\frac{(N+1)\cdot N}{2}\right) = O(N^2)$. As for the space complexity, if there is no overlapping part, the space needed is N, so the space complexity is O(N), which is the worst; the best space complexity occurs when all the time lines have overlapping part and are combined together, so the space complexity in this circumstances is O(1). And the average space complexity is $O\left(\frac{(1+2+3\ldots+N)}{N}\right) = O\left(\frac{(1+N)\cdot N}{2N}\right) = O(N)$.

4.1.2 Algorithm 2

Use quick sort in the first algorithm and we can improve the time complexity to O(NlogN), and the space complexity is as follows. When every running of the sorting divide the array into two parts with same length, the best space complexity is O(logN), but when every running of the sort generated only 1 array, the worst space complexity is O(N). The quick sort algorithm is degraded into bubble sort.

4.2 Comments

There still some parts to be improved in order to save time and space. First of all, there is no array to record the answer, so when the input computation tasks have the same time length, especially when we input 80,000 tasks with exactly the same length, 00:00:01, for example, if the array store the answer, so after the first compute, when reading the same length “00:00:01” for the next time, we don’t have to use the algorithm introduced earlier to calculate, we just need to find the answer in the answer sheet, which will only take O(N) time complexity.

Appendix: Source Code (in C)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
typedef struct TimeNode *time;
struct TimeNode;

This structure is defined to record the open / close time of the server.
What deserves your attention is that
the information of the hour, minute & the second are stored in integer form.
*/
typedef struct TimeNode time;
struct TimeNode
{
    int hour, minute, second;
};

/*
typedef struct TimeLine *timeline;
struct TimeLine;

This structure is used to store the start and end time of a server.
Define the pointer TimeLine* as timeline to point to the next data of
the start and end time of a server.
In this part, the name of the server is not that important,
because what we only care about is the length of each computing task
and whether overlapping part exists between two different records.
So, the name of the server is unnecessary to record.

The start time is hour1, minute1, second1,
while the second time is hour2, minute2, second2.
*/
```

```
typedef struct TimeLine *timeline;
struct TimeLine
{
    int hour1, minute1, second1, hour2, minute2, second2;
    timeline next;
};

/*
typedef struct ServerData *Server;
struct ServerData;

This structure is designed to store the information of each server,
containing the name of the server, server_number[7], the time_point,
which has been discussed earlier, and the integer count_time,
which is to count the number of the time_point.
For example,
if we input "za3q625 11:42:01; za3q625 06:30:50; za3q625 23:55:00",
the value of the count_time of the point "za3q625" is 3.
*/
typedef struct ServerData Server;
struct ServerData
{
    /*
    The number of the server;
    */
    char server_number[7];

    time time_point[20000];
    int count_time;
};

/*
typedef struct top *list;
struct top;

This structure is used to store all the information of the input.
And that's why it is called "top".
The array in Server type is used to store the information of the servers one by one.
The integer count_server represents the number of the server.
*/
typedef struct top *list;
struct top
{
    Server server_info[10000];
```

```
    int count_server;
};

/*
The structure compute_list is used to store the task
after all the information is stored in the structure list.
*/
typedef struct compute_list task;
struct compute_list
{
    timeline compute_task;
    int count_task;
};

/*
Initialize the list,
the number of the server is initialized as 0.
*/
list init_list(void);

/*
Read N queries of the server infomation with format like "xxxxxxx hh:mm:ss",
then store them by insertion sort or quick sort.
*/
list Read(int n);

/*
The function AddTheData is designed to add the time point to the type task.
The main algorithm is to use a double-while-loop
in order to read all the time point.
*/
task AddTheData(list general_list);

/*
The function is able to decide
whether the two input strings are the same
and return 1 if they are they are the same, return 0 if not.
Meanwhile, the function can add the time point after the existed record
if the strings are the same,
and because the same strings represent the same server,
the data of the same server should be stored together.
*/
int string_compare(list a, char b[], int hour, int minute, int second);
```

```
/*
This function has the capability
to sort the time point in increasing order by insertion sort,
or quick sort.
*/
void add(list a, int hour, int minute, int second, int n);

/*
Compute the total seconds of a time line
and return the total value in integer type.
*/
int compute_second(int hour1, int minute1, int second1, int hour2, int minute2, int second2);

/*
The function compares the value of two time,
h1:m1:s1 & h2:m2:s2,
if h1:m1:s1 > h2:m2:s2, return 1;
else, return 0.
*/
int compare_time(int hour1, int minute1, int second1, int hour2, int minute2, int second2);

/*
The function insert the time point to the list.
If the list is empty, add the time point to the server_info[0]
*/
void insert(list a, char name[], int hour, int minute, int second);
int add_time(timeline a, time t1, time t2, int count);

int main ()
{
    /*
    Each input file contains one test case.
    Each case starts with two positive integers N (≤2×10^4),
    the number of server records, and K (≤8×10^4),
    the number of queries.
    */
    int n, k, i, j;
    scanf("%d %d", &n, &k);

    /*
    Task 1:
    Read the input
    and return the general_list.
    */
```

```
    list general_list = Read(n);

    /*
    Task 2:
    Combine the time line
    and store them in the array
    in task type.
    */
    task time_line = AddTheData(general_list);

    /*
    Initialize the integer array time_list,
    which is used to store the time lines.
    */
    int time_list[time_line.count_task];

    /*
    Pointer tmp is used to traverse the linked list time_line.compute_task.
    */
    timeline tmp = time_line.compute_task;

    /*
    The integer represents
    the length of the integer array time_list.
    */
    int index = 0;
    while (tmp)
    {
        time_list[index++] = compute_second(tmp->hour1, tmp->minute1, tmp->second1,
tmp->hour2, tmp->minute2, tmp->second2);
        tmp = tmp->next;
    }

    /*
    Find the maximum computation task you could run
    in the integer array time_list.
    */
    int max = time_list[0];
    for (i = 1; i < index; i++)
        max = (time_list[i] > max) ? time_list[i] : max;

    /*
    The integer array answer[k] is used to store
    the answer, in other word, the total number
```

```
	of valid time points
	for starting the given computation task.
	*/
	int answer[k];
	for (i = 0; i < k; i++)
	{
		int hour, minute, second;
		scanf("%d:%d:%d", &hour, &minute, &second);
		int t = compute_second(0, 0, 0, hour, minute, second);
		int count_total_time = 0;
		for (j = 0; j < index; j++)
		{
			if (time_list[j] >= t)
			{
				count_total_time += time_list[j] - t + 1;
			}
		}
		answer[i] = count_total_time;
	}
	/*
	Output the longest computation task you could run.
	*/
	printf("\n%d\n", max);
	for (i = 0; i < k; i++)
	{
		/*
		Output the total number
		of valid time points
		for starting the given computation task, one by one.
		*/
		printf("%d\n", answer[i]);
	}
	printf("\n");

	/*
	Give the system a pause.
	*/
	system("pause");
	return 0;
}

list init_list(void)
{
	/*
```

```
    Init a new list a with function malloc.
    */
    list a = (list)malloc(sizeof(struct top));
    a->count_server = 0;
    return a;
}

int add_time(timeline a, time t1, time t2, int count)
{
    int i;
    int hour1, minute1, second1, hour2, minute2, second2;
    hour1 = t1.hour, minute1 = t1.minute, second1 = t1.second;
    hour2 = t2.hour, minute2 = t2.minute, second2 = t2.second;

    /*
    If count = 0, which means the timeline is NULL,
    so the input time point is the first one,
    we need to add it to the head of the timeline 1.
    */
    if (!count)
    {
        a->hour1 = hour1;
        a->minute1 = minute1;
        a->second1 = second1;
        a->hour2 = hour2;
        a->minute2 = minute2;
        a->second2 = second2;
        count++;
        a->next = NULL;
    }
    /*
    When the timeline a is not empty.
    */
    else
    {
        timeline obj = (timeline)malloc(sizeof (struct TimeLine));
        timeline tmp = a, s = a;
        obj->hour1 = hour1, obj->minute1 = minute1, obj->second1 = second1;
        obj->hour2 = hour2, obj->minute2 = minute2, obj->second2 = second2;
        obj->next = NULL;
        while (s->next) s = s->next;

        s->next = obj;
        count++;
```

```
        while (tmp->next)
        {
            if (!compare_time(tmp->hour1, tmp->minute1, tmp->second1, hour1, minute1, second1) && compare_time(tmp->hour2, tmp->minute2, tmp->second2, hour2, minute2, second2))
            {
                /*
                If the new timeline
                is the subset of the new timeline,
                just ignore the new one.
                */
                obj = NULL;
                count--;
                break;
            }
            else if (compare_time(tmp->hour1, tmp->minute1, tmp->second1, hour2, minute2, second2) || !compare_time(tmp->hour2, tmp->minute2, tmp->second2, hour1, minute1, second1))
            {
                /*
                If the two timelines have no overlapping parts,
                we just need to let the pointer tmp
                points to tmp->next,
                and then continue the while loop.
                */
                tmp = tmp->next;
                continue;
            }
            else
            {   /*
                If there is overlapping part between the two time points,
                the further discussion is needed.
                */

                /*
                If the old time line
                is the subset of the new subset,
                what we only need to do
                is to delete the old time line
                later.
                */
                if (compare_time(tmp->hour1, tmp->minute1, tmp->second1, hour1, minute1, second1) && !compare_time(tmp->hour2, tmp->minute2, tmp->second2, hour2,
```

```
minute2, second2));

                    /*
                    If there is overlapping part,
                    the value of hour1, minute1, second1
                    or hour2, minue2, second2,
                    will change.
                    */
                    else if (!compare_time(tmp->hour1, tmp->minute1, tmp->second1, hour2, minute2, second2) && compare_time(tmp->hour2, tmp->minute2, tmp->second2, hour2, minute2, second2))
                    {
                        obj->hour2 = tmp->hour2;
                        obj->minute2 = tmp->minute2;
                        obj->second2 = tmp->second2;
                    }
                    else if (!compare_time(tmp->hour1, tmp->minute1, tmp->second1, hour1, minute1, second1) && compare_time(tmp->hour2, tmp->minute2, tmp->second2, hour1, minute1, second1))
                    {
                        obj->hour1 = tmp->hour1;
                        obj->minute1 = tmp->minute1;
                        obj->second1 = tmp->second1;
                    }
                    else
                    {
                        tmp = tmp->next;
                        continue;
                    }
                    if (tmp == a)
                        a = a->next;
                    else
                    {
                        timeline load = (timeline)malloc(sizeof (struct TimeLine));
                        load = a;
                        while(1)
                        {
                            if (load->next == tmp) break;
                            load = load->next;
                        }
                        load->next = load->next->next;
                    }
                    count--;
                    tmp = tmp->next;
```

```
                }
            }
        }
        return count;
}

list Read(int n)
{
        list general_list = init_list();
        int i;
        for (i = 0; i < n; i++)
        {
                char name[7];
                int hour, minute, second;
                scanf("%s %d:%d:%d", name, &hour, &minute, &second);
                /*
                If the server is the first time to read,
                then we need to constructe a new space
                to store the information of the server.
                */
                if (!string_compare(general_list, name, hour, minute, second))
                        insert(general_list, name, hour, minute, second);
        }
        return general_list;
}

task AddTheData(list general_list)
{
        int i, j;
        task time_line;
        time_line.compute_task = (timeline)malloc(sizeof (struct TimeLine));
        time_line.count_task = 0;

        /*
        In the double-while loop,
        traverse the the general list,
        store the total time line in time_line.count_task.
        */
        for (i = 0; i < general_list->count_server; i++)
        {
                for (j = 0; (j+1) < general_list->server_info[i].count_time; j += 2)
                {
                        time_line.count_task = add_time(time_line.compute_task,
general_list->server_info[i].time_point[j], general_list->server_info[i].time_point[j+1],
```

```
time_line.count_task);
            }
        }
        return time_line;
}

int string_compare(list a, char b[], int hour, int minute, int second)
{
        if (!a->count_server) return 0;
        int i, flag = 0;
        for (i = 0; i < a->count_server; i++)
        {
                if (!strcmp(b, a->server_info[i].server_number))
                {
                        flag = 1;
                        /*
                        Add the open / close time of the server which has already exists
                        */
                        add(a, hour, minute ,second, i);
                        break;
                }
        }
        if (flag) return 1;
        return 0;
}

void add(list a, int hour, int minute, int second, int n)
{
        int i;
        if (a->server_info[n].count_time != 0)
        {
                for (i = a->server_info[n].count_time-1; i >= 0; i--)
                {
                        if (compare_time(a->server_info[n].time_point[i].hour,
a->server_info[n].time_point[i].minute, a->server_info[n].time_point[i].second, hour, minute,
second))
                        {
                                a->server_info[n].time_point[i+1] = a->server_info[n].time_point[i];
                        }
                        else break;
                }
        }
        a->server_info[n].time_point[i+1].hour = hour;
        a->server_info[n].time_point[i+1].minute = minute;
```

```
	a->server_info[n].time_point[i+1].second = second;
	a->server_info[n].count_time++;
}

/*
Compute the total seconds of a time line and return the total value in integer type.
*/
int compute_second(int hour1, int minute1, int second1, int hour2, int minute2, int second2)
{
	/*
	The total length of the time line in second
	is hour*3600+minute*60+second.
	*/
	return (hour2*3600+minute2*60+second2)-(hour1*3600+minute1*60+second1);
}

/*
The function compares the value of two time,
h1:m1:s1 & h2:m2:s2.
If h1:m1:s1 > h2:m2:s2, return 1;
else, return 0.
*/
int compare_time(int hour1, int minute1, int second1, int hour2, int minute2, int second2)
{
	/*
	If timeline1 > timeline2, return 1;
	else return 0.
	*/
	if (hour1 > hour2)
		return 1;
	else if (hour1 < hour2)
		return 0;
	else if (hour1 == hour2)
	{
		if (minute1 > minute2)
			return 1;
		else if (minute1 < minute2)
			return 0;
		else if (minute1 == minute2)
		{
			if (second1 > second2)
				return 1;
			else return 0;
		}
```

```
        }
}

void insert(list a, char name[], int hour, int minute, int second)
{
        /*
        The function insert the time point to the list.
        If the list is empty, add the time point to the server_info[0]:
        */
        int i;
        if (!a->count_server)
        {
                a->server_info[0].count_time = 0;
                a->count_server = 0;
                for (i = 0; i < 7; i++)
                        a->server_info[0].server_number[i] = name[i];
                a->server_info[0].time_point[0].hour = hour;
                a->server_info[0].time_point[0].minute = minute;
                a->server_info[0].time_point[0].second = second;
                a->server_info[0].count_time++;
        }
        else
        {
                a->server_info[a->count_server].count_time = 0;
                for (i = 0; i < 7; i++)
                        a->server_info[a->count_server].server_number[i] = name[i];
                a->server_info[a->count_server].time_point[0].hour = hour;
                a->server_info[a->count_server].time_point[0].minute = minute;
                a->server_info[a->count_server].time_point[0].second = second;
                a->server_info[a->count_server].count_time++;
        }
        a->count_server++;
}
```

Declaration

I hereby declare that all the work done in this project titled "Project 3. Arrangement of Computation Tasks" is of my independent effort.